CS 2710 / ISSP 2610

Planning

Planning

- What is classical planning?
- Approaches
  - STRIPS/PDDL
  - State-Space Search
  - Planning Graphs
  - Satisfiability
  - Situation Calculus
  - Partially Ordered Plans

Planning problem

- Find a **sequence of actions** that achieves a given **goal** when executed from a given **initial world state**. That is, given
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,

  compute a plan, which is
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of goals to be achieved

Planning as Search-Based Problem Solving?

- Imagine a supermarket shopping scenario using search-based problem solving:
  - Goal: buy milk and bananas
  - Operator: buy <obj>
  - Heuristic function: does <obj> = milk or bananas?

- The operator would be instantiated with all possible objects that can be bought! Then the heuristic function would evaluate each instantiation. This is essentially a guessing game!

Least Commitment

- Or… suppose you haven't decided where to go shopping.
  - Goal: buy milk and bananas
  - Operators: go_to<store>, buy<obj,store>
  - You can get milk at the convenience store, the dairy, or the supermarket.
  - You can only get bananas at the supermarket.
- If you decide where to buy milk first (say, at the convenience store), then you will either:
  - have to backtrack, or
  - have to go to more than one store!
- Planners need to be more flexible

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search can proceed through *plan space* rather than *state space* (though there are also state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Typical assumptions

- Atomic time: Each action is indivisible
- No concurrent actions are allowed (though actions do not need to be ordered with respect to each other in the plan)
- Deterministic actions: The result of actions are completely determined—there is no uncertainty in their effects
- Agent is the sole cause of change in the world
- Agent is omniscient: Has complete knowledge of the state of the world
- Closed World Assumption: everything known to be true in the world is included in the state description. Anything not listed is false.

Blocks world

The **blocks world** is a micro-world that consists of a table, a set of blocks and a robot hand.

Some domain constraints:

- Only one block can be on another block
- Any number of blocks can be on the table
- The hand can only hold one block

Typical representation:

ontable(a)

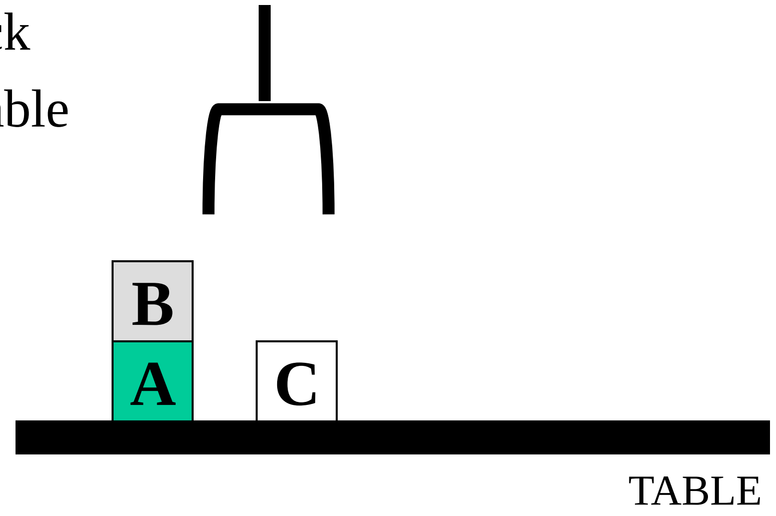
ontable(c)

on(b,a)

handempty

clear(b)

clear(c)



Situation calculus planning

- Intuition: Represent the planning problem using first-order logic
  - Situation calculus lets us reason about changes in the world
  - Use theorem proving to “prove” that a particular sequence of actions, when applied to the situation characterizing the world state, will lead to a desired result

Situation Calculus

- Logic for reasoning about changes in the state of the world
- The world is described by
  - Sequences of situations of the current state
  - Changes from one situation to another are caused by actions
- The situation calculus allows us to
  - Describe the initial state and a goal state
  - Build the KB that describes the effect of actions (operators)
  - Prove that the KB and the initial state lead to a goal state
  - Extracts a plan as side-effect of the proof

Situation Calculus Ontology

- Actions: terms, such as “forward” and “turn(right))”
- Situations: terms; initial situation s0 and all situations that are generated by applying an action to a situation. result(a,s) names the situation resulting when action a is done in situation s.

Situation Calculus Ontology continued

- Fluents: functions and predicates that vary from one situation to the next. By convention, the situation is the last argument of the fluent. ~holding(robot,gold,s0)
- Atemporal or eternal predicates and functions do not change from situation to situation. gold(g1). lastName(wumpus,smith). adjacent(livingRoom,kitchen).

Frame Problem

- We run into the frame problem
- Effect axioms say what changes, but don't say what stays the same
- A real problem, because (in a non-toy domain), each action affects only a tiny fraction of all fluents

- We will return to situation calculus later…

Basic representations for planning

- Classic approach first used in the **STRIPS** planner circa 1970
- States represented as a conjunction of ground literals
  - at(Home) ^ ~have(Milk) ^ ~have(bananas) ...
- Goals are conjunctions of literals, but may have variables which are assumed to be existentially quantified
  - at(?x) ^ have(Milk) ^ have(bananas) ...
- Do not need to fully specify state
  - Non-specified either don't-care or assumed false
  - Represent many cases in small storage
  - Often only represent changes in state rather than entire situation
- Unlike theorem prover, not seeking whether the goal is true, but is there a sequence of actions to attain it

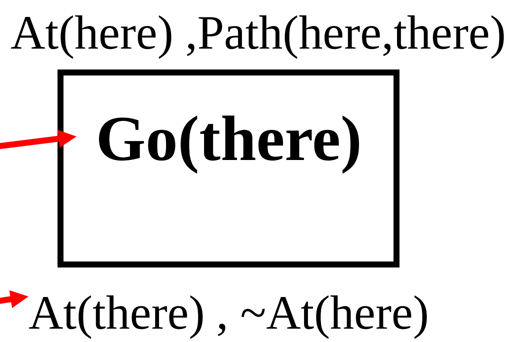
Operator/action representation

- Operators contain three components:
  - **Action description**
  - **Precondition** - conjunction of positive literals
  - **Effect** - conjunction of positive or negative literals which describe how situation changes when operator is applied
- Example:

  Op[Action: Go(there),

  Precond: At(here) ^ Path(here,there),

  Effect: At(there) ^ ~At(here)]



- All variables are universally quantified
- Situation variables are implicit
  - preconditions must be true in the state immediately before operator is applied; effects are true immediately after

Blocks world operators

- Here are the classic basic operations for the blocks world:
  - stack(X,Y): put block X on block Y
  - unstack(X,Y): remove block X from block Y
  - pickup(X): pickup block X from the table
  - putdown(X): put block X on the table
- Each will be represented by
  - a list of preconditions
  - a list of new facts to be added (add-effects)
  - a list of facts to be removed (delete-effects)
  - optionally, a set of (simple) variable constraints
- For example:

  preconditions(stack(X,Y), [holding(X),clear(Y)])

  deletes(stack(X,Y), [holding(X),clear(Y)]).

  adds(stack(X,Y), [handempty,on(X,Y),clear(X)])

  constraints(stack(X,Y), [X ~= Y,Y ~= table,X ~= table])

Blocks world operators II

```
operator(stack(X,Y),
      Precond [holding(X),clear(Y)],
      Add [handempty,on(X,Y),clear(X)],
      Delete [holding(X),clear(Y)],
      Constr [X ~=Y,Y ~=table,X ~= table]).
```

```
operator(unstack(X,Y),
      [on(X,Y), clear(X), handempty],
      [holding(X),clear(Y)],
      [handempty,clear(X),on(X,Y)],
      [X ~= Y,Y ~= table, X ~= table]).
```

```
operator(pickup(X),
      [ontable(X), clear(X), handempty],
      [holding(X)],
      [ontable(X),clear(X),handempty],
      [X ~= table]).
```

```
operator(putdown(X),
      [holding(X)],
      [ontable(X),handempty,clear(X)],
      [holding(X)],
      [X ~= table]).
```

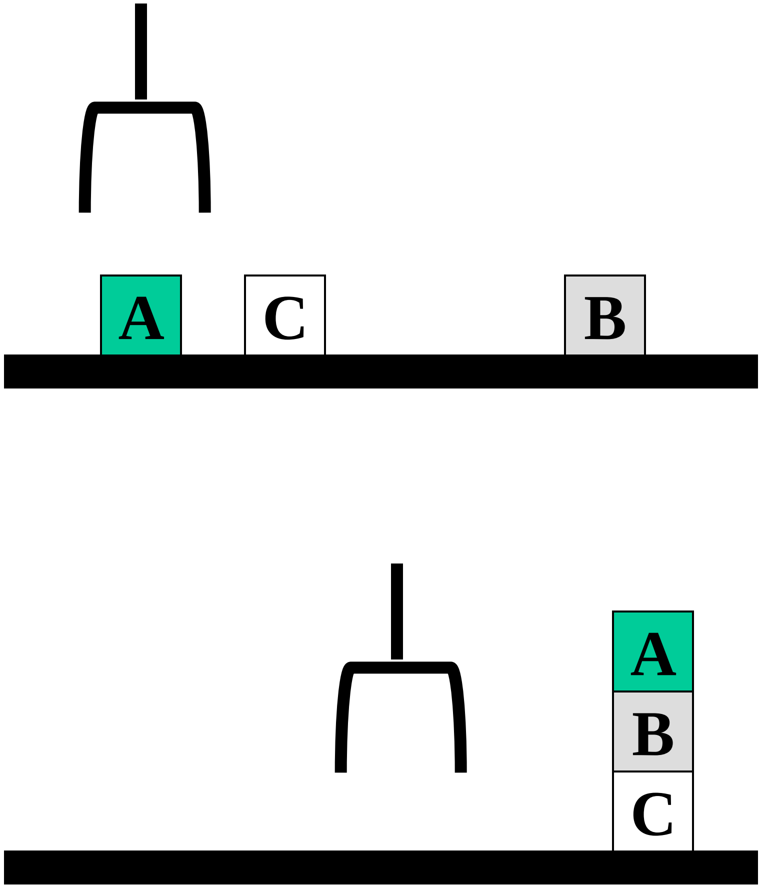
Typical BW planning problem

Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(b,c)
- on(a,b)
- ontable(c)



A plan:

- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

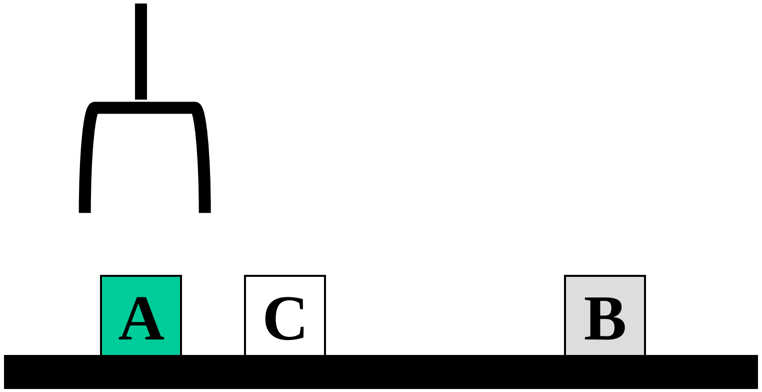
Another BW planning problem

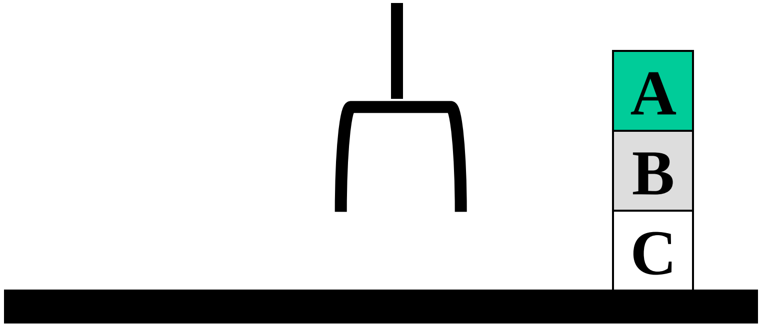
Initial state:

- clear(a)
- clear(b)
- clear(c)
- ontable(a)
- ontable(b)
- ontable(c)
- handempty

Goal:

- on(a,b)
- on(b,c)
- ontable(c)





A plan:

- pickup(a)
- stack(a,b)
- unstack(a,b)
- putdown(a)
- pickup(b)
- stack(b,c)
- pickup(a)
- stack(a,b)

Goal interaction

- Simple planning algorithms assume that the goals to be achieved are independent
  - Each can be solved separately and then the solutions concatenated
- This planning problem, called the "Sussman Anomaly," is the classic example of the goal interaction problem:
  - Solving on(A,B) first (by doing unstack(C,A), stack(A,B) will be undone when solving the second goal on(B,C) (by doing unstack(A,B), stack(B,C)).
  - Solving on(B,C) first will be undone when solving on(A,B)
- Classic STRIPS could not handle this, although minor modifications can get it to do simple cases

Initial state

Goal state

State-space planning

- We initially have a space of situations (where you are, what you have, etc.)
- The plan is a solution found by “searching” through the situations to get to the goal
- A **progression planner** searches forward from initial state to goal state
- A **regression planner** searches backward from the goal
  - This works if operators have enough information to go both ways
  - Ideally this leads to reduced branching –you are only considering things that are relevant to the goal

Planning Graphs

- Construct a graph that encodes constraints on possible plans
- Use this "planning graph" to constrain search for a valid plan:
  - If valid plan exists, it's a subgraph of the planning graph
  - Can also provide heuristics for search algorithms
- Planning graph can be built for each problem in polynomial time

Problem handled by GraphPlan*

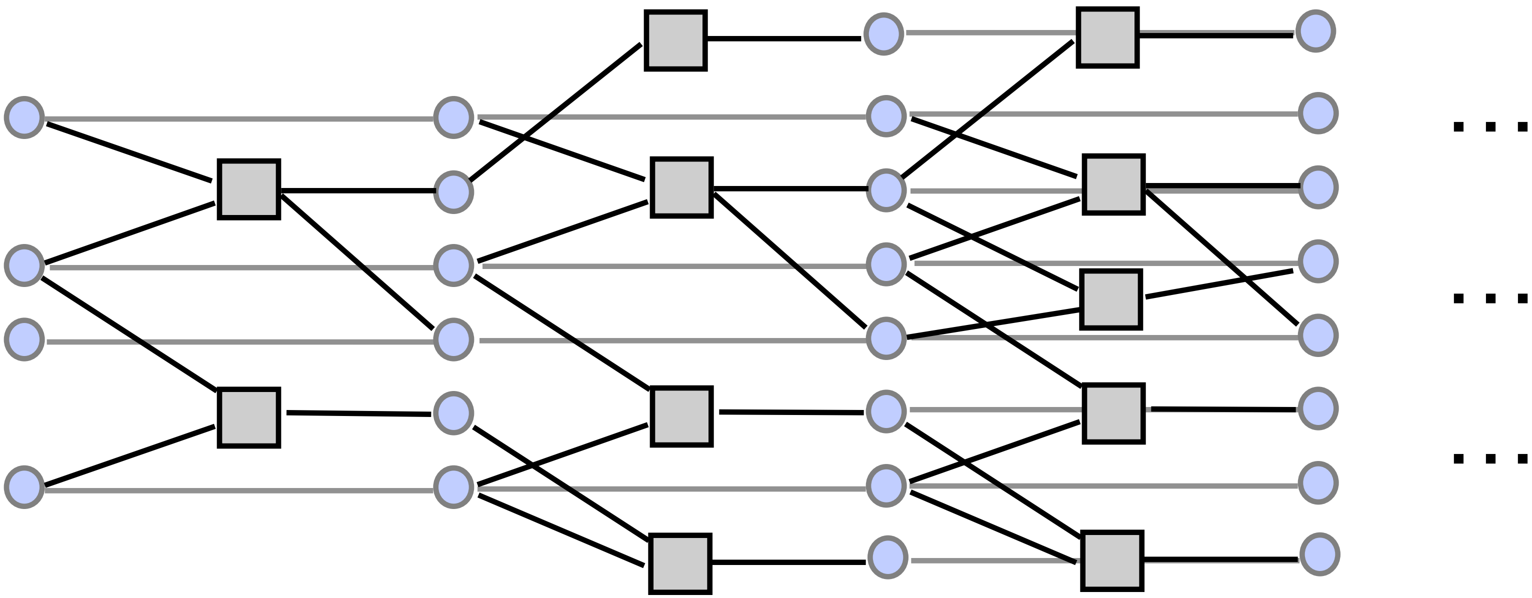
- Pure STRIPS operators:
  - conjunctive preconditions
  - no negated preconditions
  - no conditional effects
  - no universal effects
- Finds "shortest parallel plan"
- Sound, complete and will terminate with failure if there is no plan.

*Version in [Blum& Furst IJCAI 95, AIJ 97],
later extended to handle all these restrictions [Koehler et al 97]

Planning graph

- Directed, leveled graph
  - 2 types of nodes:
    - Proposition: P
    - Action: A
  - 3 types of edges (between levels)
    - Precondition: P -> A
    - Add: A -> P
    - Delete: A -> P
- Proposition and action levels alternate
- Action level includes actions whose preconditions are satisfied in previous level plus no-op actions (to solve frame problem).

Planning graph



Constructing the planning graph

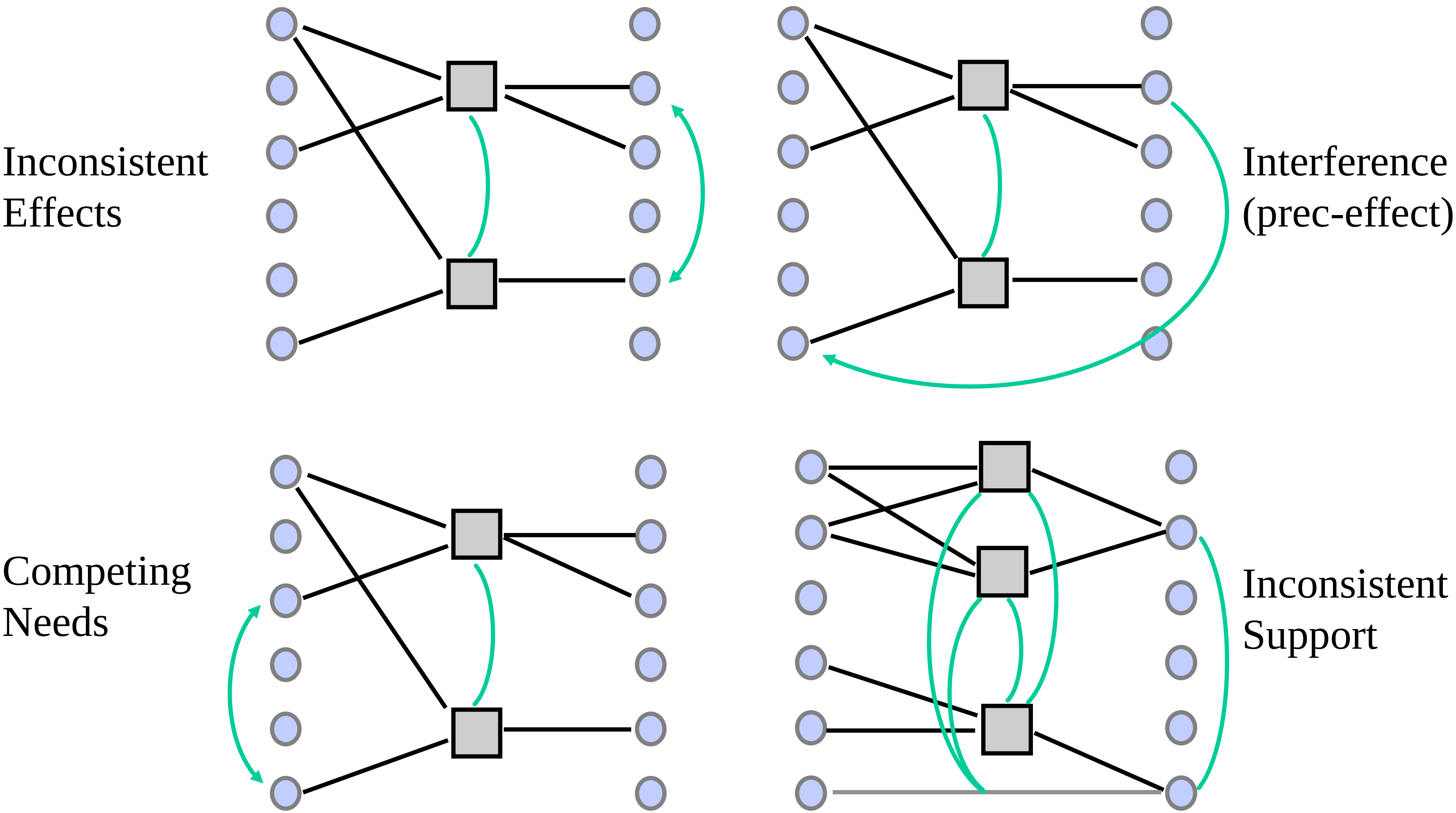
- Level P_1: all literals from the initial state
- Add an action in level A_i if all its preconditions are present in level P_i
- Add a precondition in level P_i if it is the effect of some action in level A_{i-1} (including no-ops)
- Maintain a set of exclusion relations to eliminate incompatible propositions and actions (thus reducing the graph size)

$$P_1\ A_1\ P_2\ A_2\ \ldots\ P_{n-1}\ A_{n-1}\ P_n$$

Mutual Exclusion relations

- Two actions (or literals) are mutually exclusive (mutex) at some stage if no valid plan could contain both.
- Two actions are mutex if:
  - Interference: one clobbers others' effect or precondition
  - Competing needs: mutex preconditions
- Two propositions are mutex if:
  - All ways of achieving them are mutex
  - They negate each other

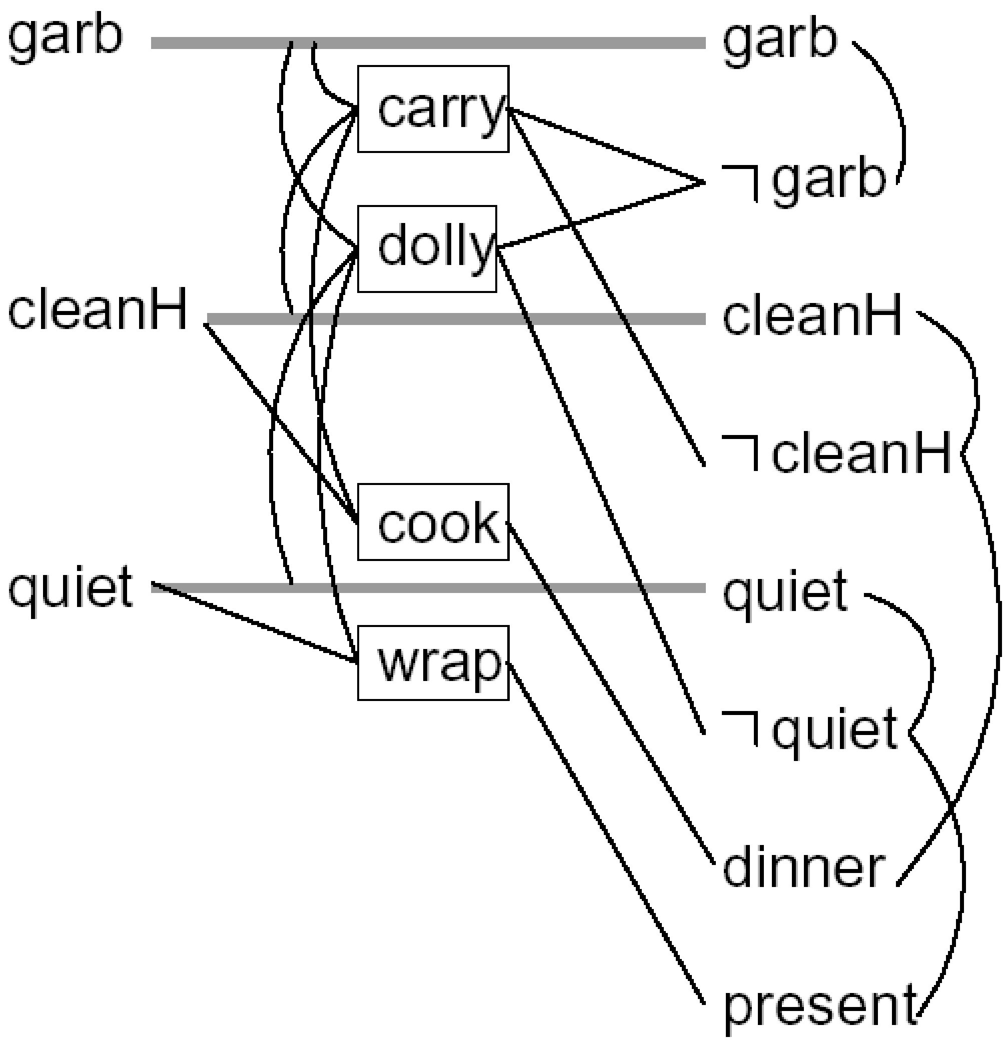
Mutual Exclusion relations



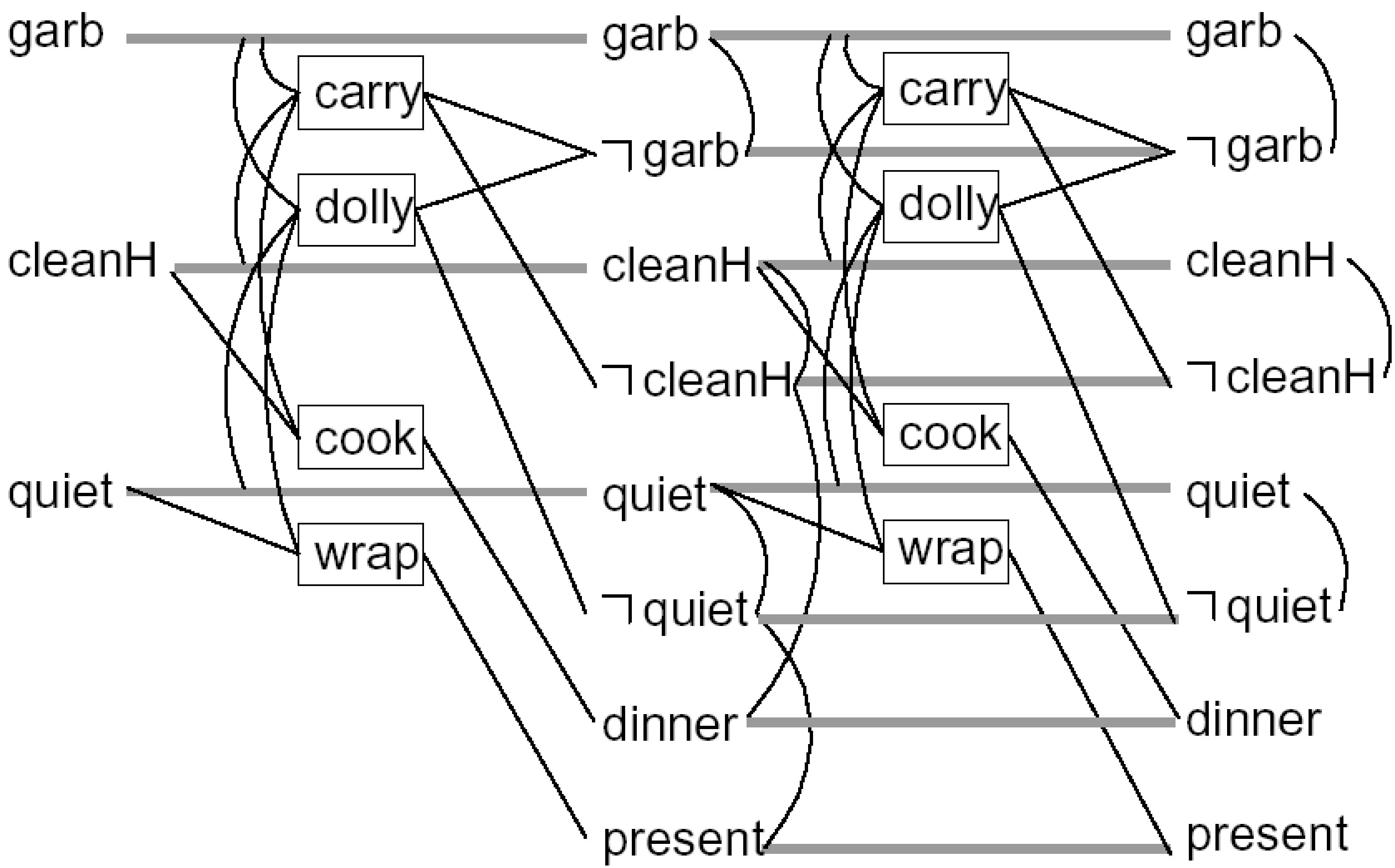
Dinner Date example

- Initial Conditions: (and (garbage) (cleanHands) (quiet))
- Goal: (and (dinner) (present) (not (garbage))
- Actions:
  - Cook :precondition (cleanHands)
    :effect (dinner)
  - Wrap :precondition (quiet)
    :effect (present)
  - Carry :precondition
    :effect (and (not (garbage)) (not (cleanHands))
  - Dolly :precondition
    :effect (and (not (garbage)) (not (quiet)))

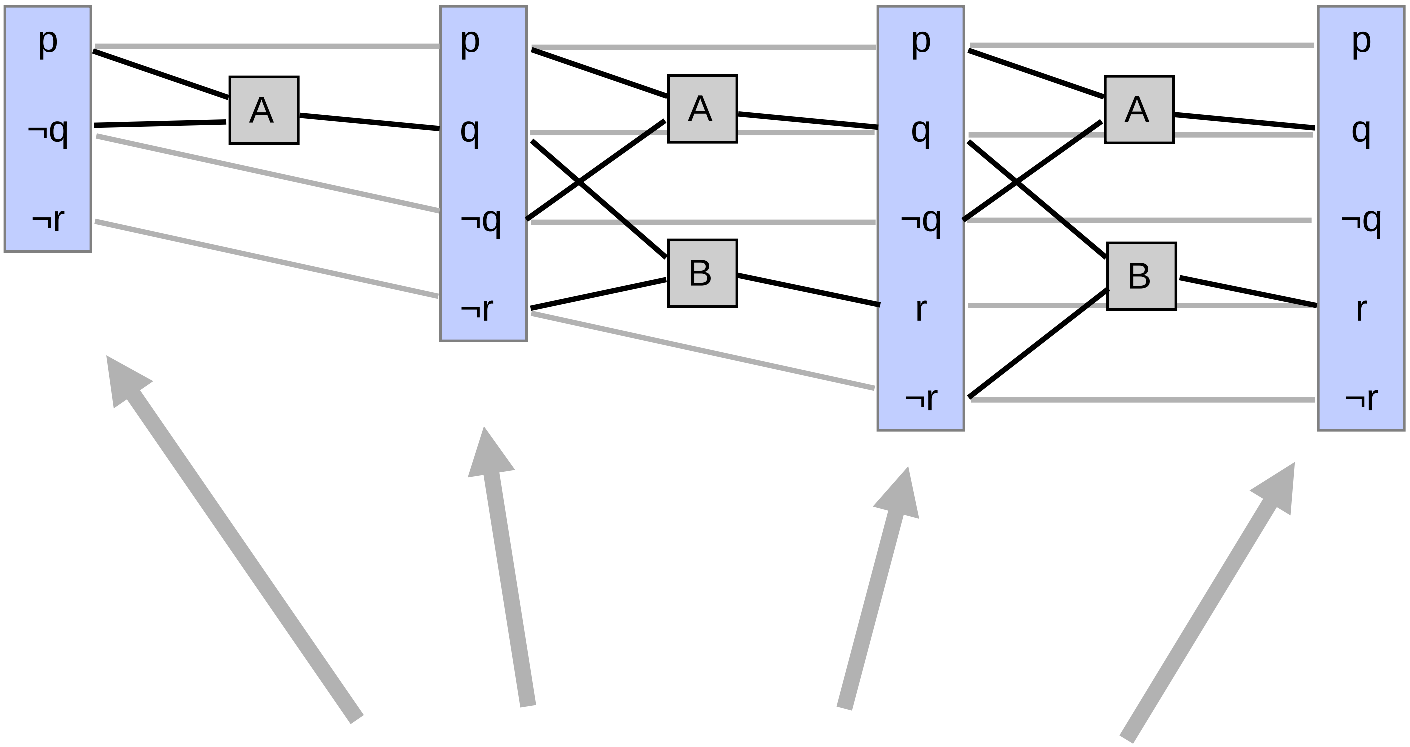
Dinner Date example



Dinner Date example



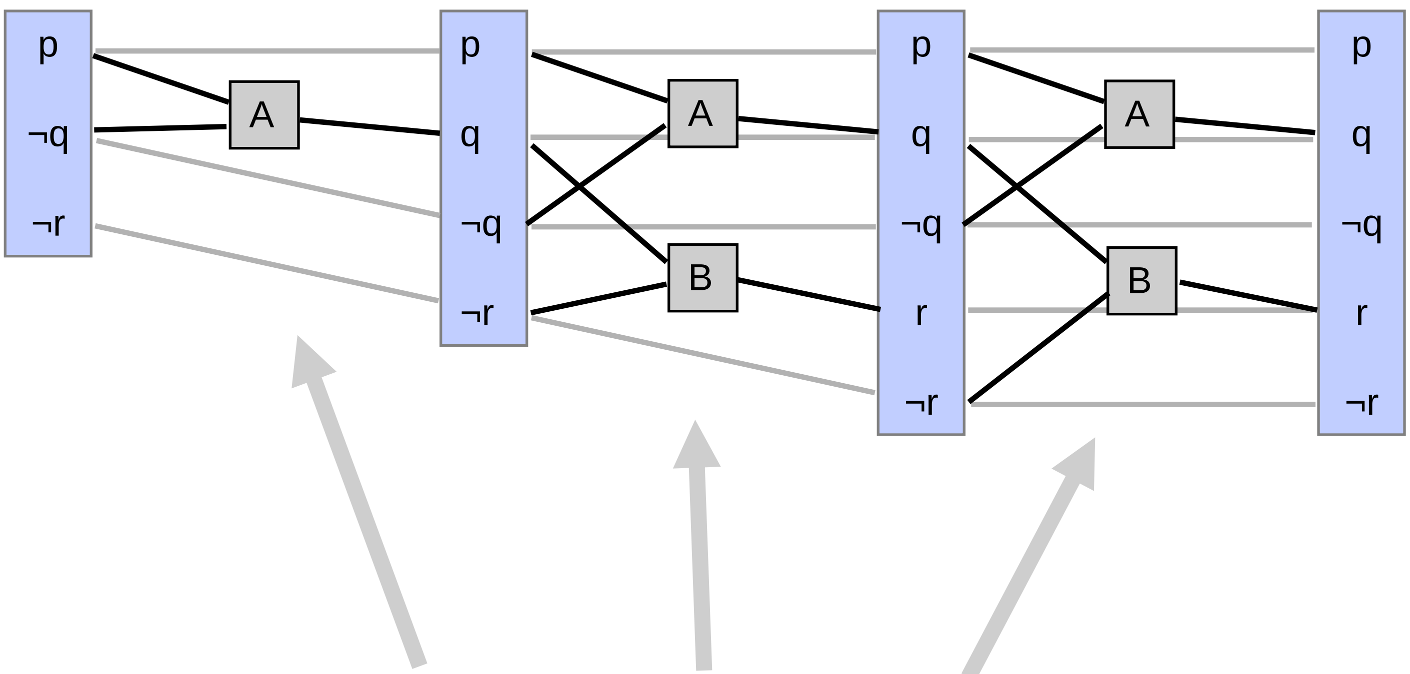
Observation 1



Propositions monotonically increase

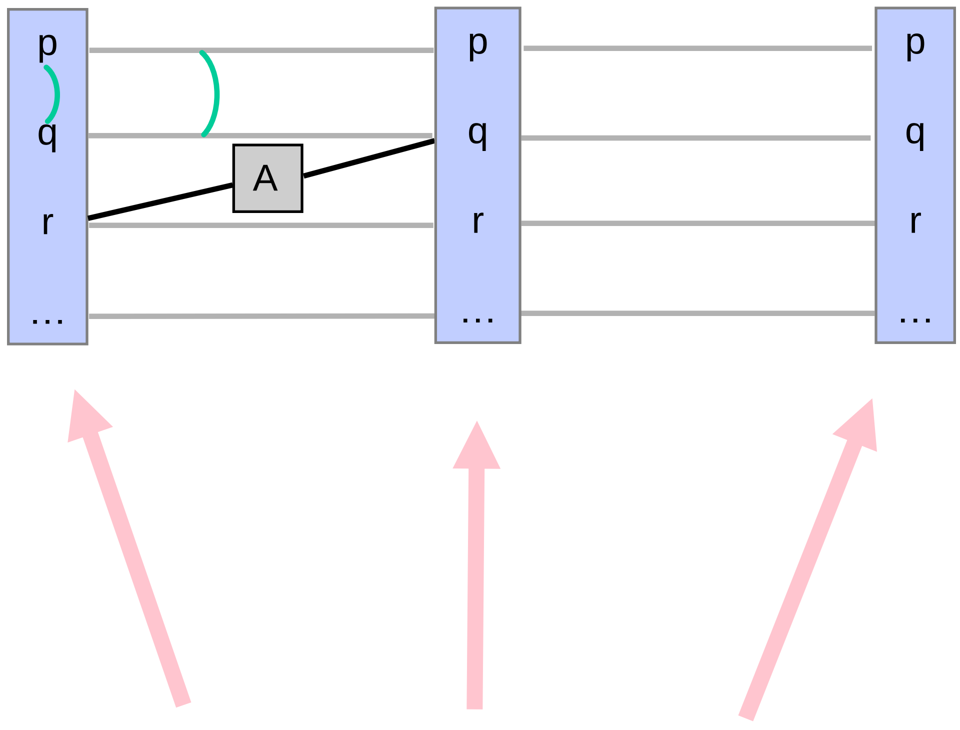
(always carried forward by no-ops)

Observation 2



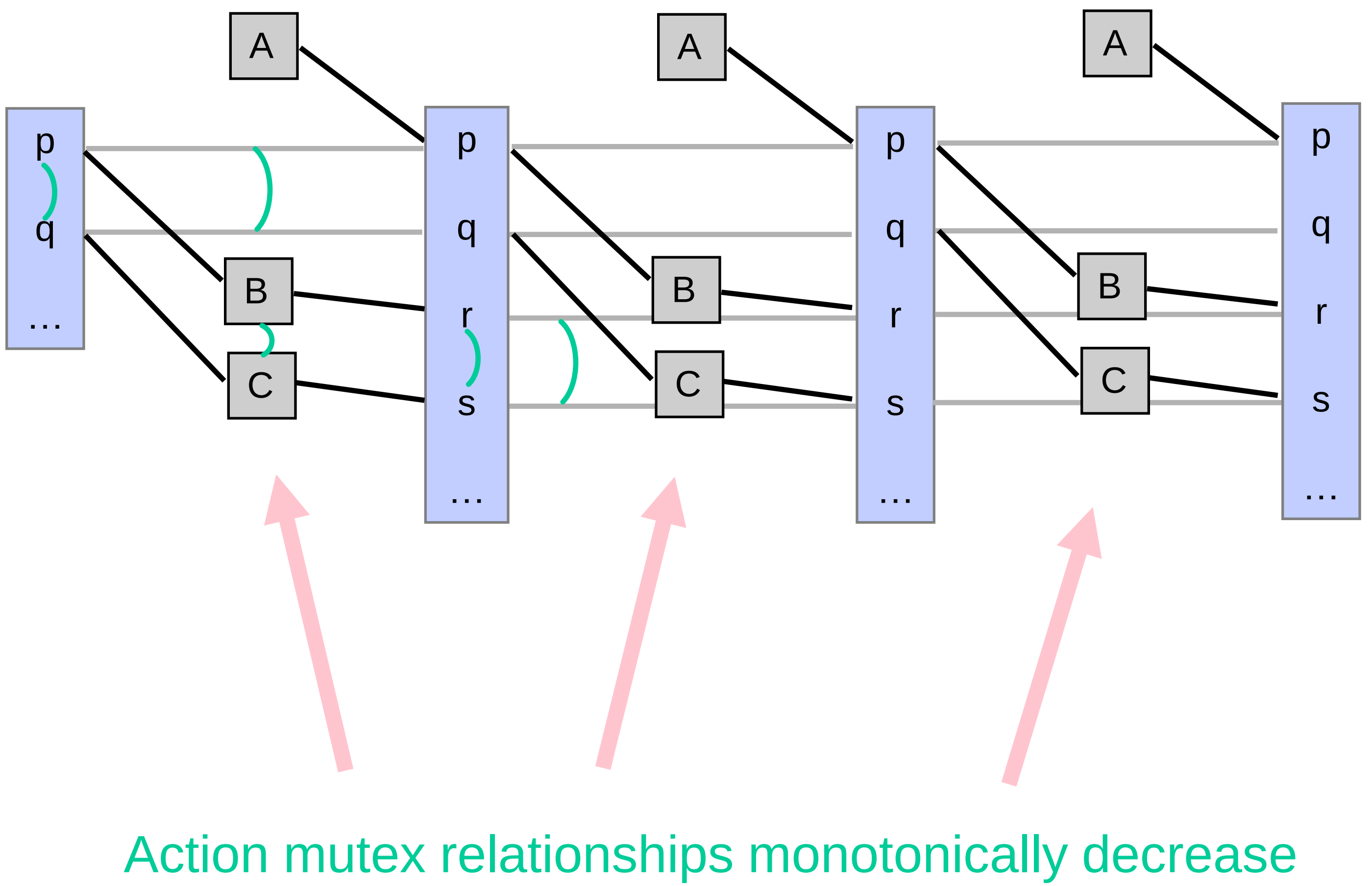
Actions monotonically increase

Observation 3



Proposition mutex relationships monotonically decrease

Observation 4



Observation 5

Planning Graph ‘levels off’.

- After some time k all levels are identical
- Because it’s a finite space, the set of literals never decreases and mutexes don’t reappear.

Valid plan

A valid plan is a planning graph where:

- Actions at the same level don’t interfere
- Each action’s preconditions are made true by the plan
- Goals are satisfied

GraphPlan algorithm

- Grow the planning graph (PG) until all goals are reachable and not mutex. (If PG levels off first, fail)
- Search the PG for a valid plan
- If non found, add a level to the PG and try again

Searching for a solution plan

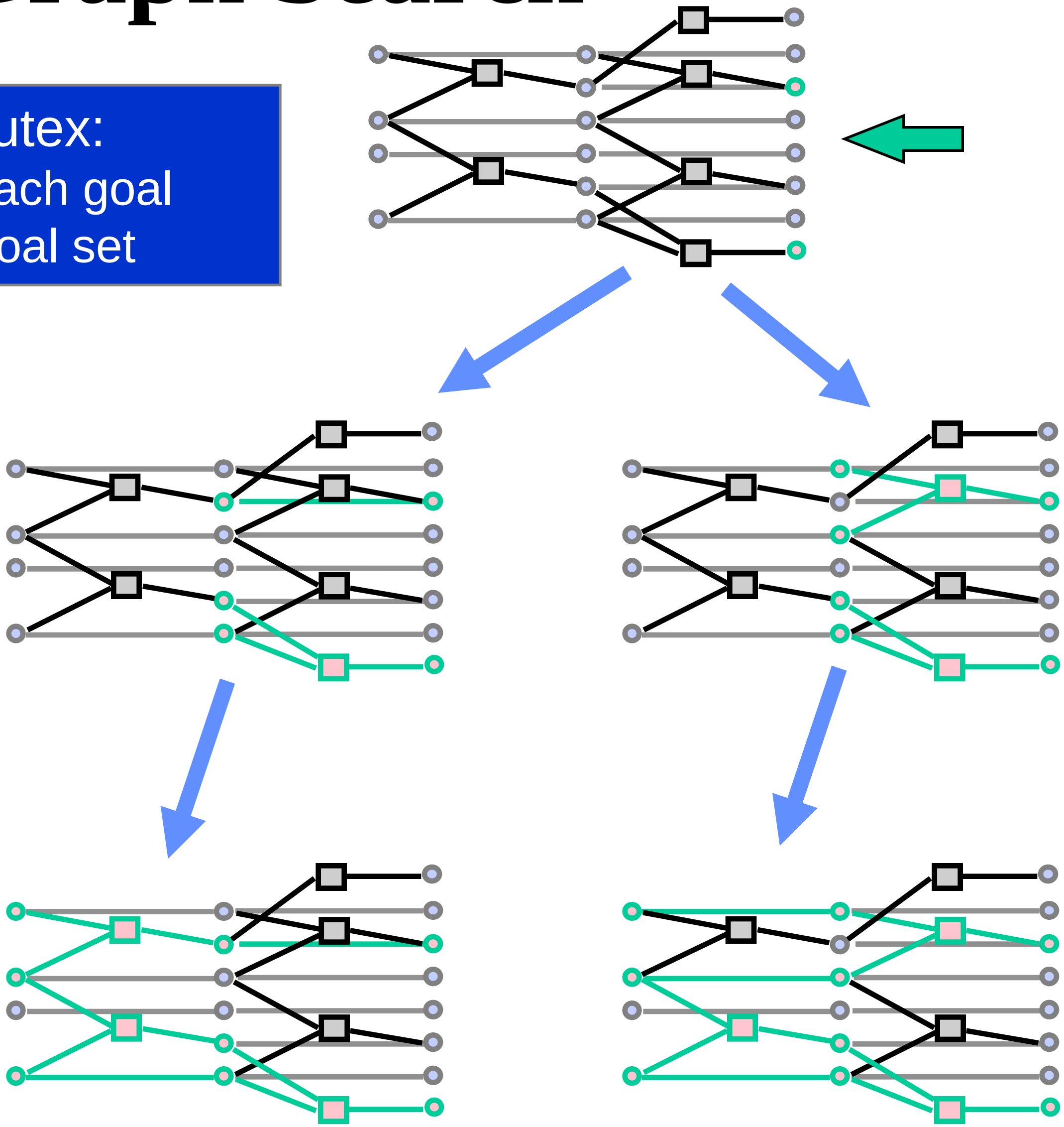
- Backward chain on the planning graph
- Achieve goals level by level
- At level k, pick a subset of non-mutex actions to achieve current goals. Their preconditions become the goals for k-1 level.
- Build goal subset by picking each goal and choosing an action to add. Use one already selected if possible. Do forward checking on remaining goals (backtrack if can't pick non-mutex action)

Plan Graph Search

If goals are present & non-mutex:

Choose action to achieve each goal

Add preconditions to next goal set



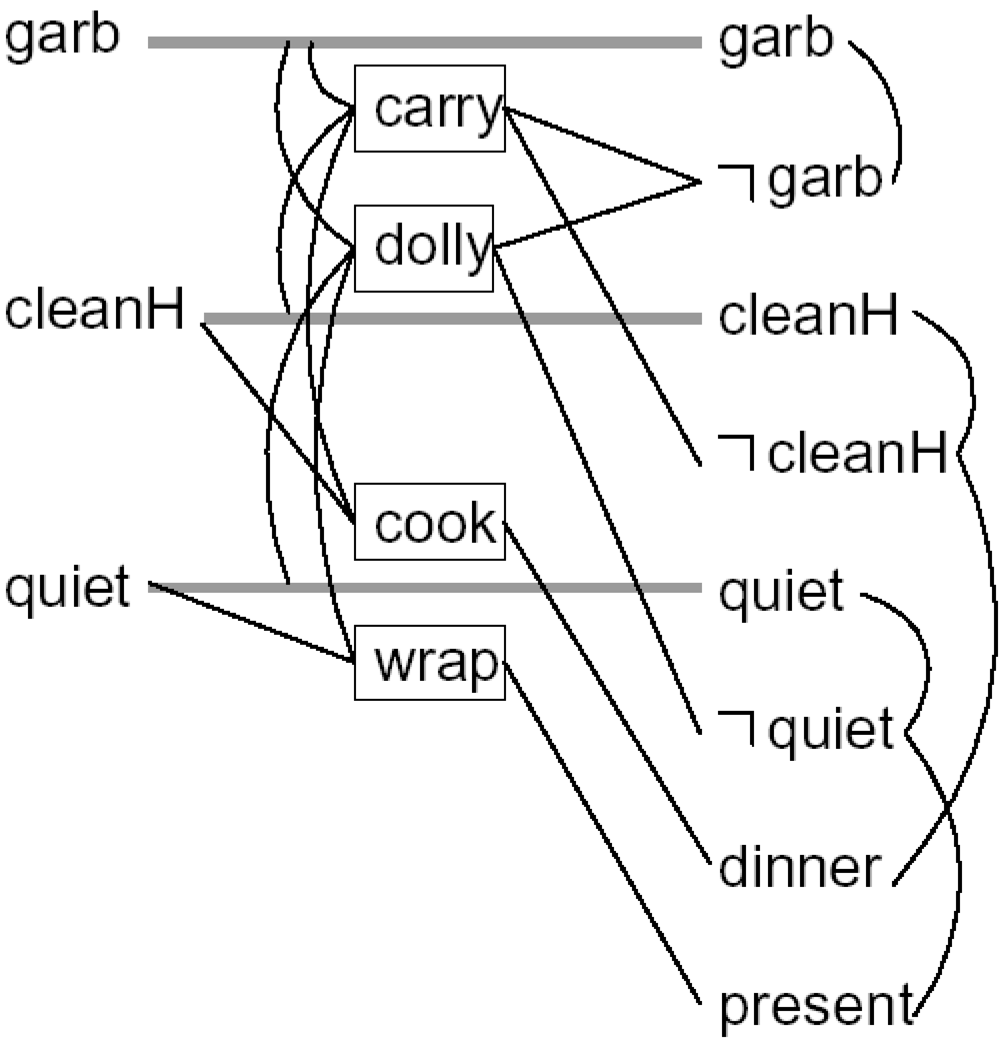
Termination for unsolvable problems

- Graphplan records (memoizes) sets of unsolvable goals:
  - U(i,t) = unsolvable goals at level i after stage t.
- More efficient: early backtracking
- Also provides necessary and sufficient conditions for termination:
  - Assume plan graph levels off at level n, stage t > n
  - If U(n, t-1) = U(n, t) then we know we're in a loop and can terminate safely.

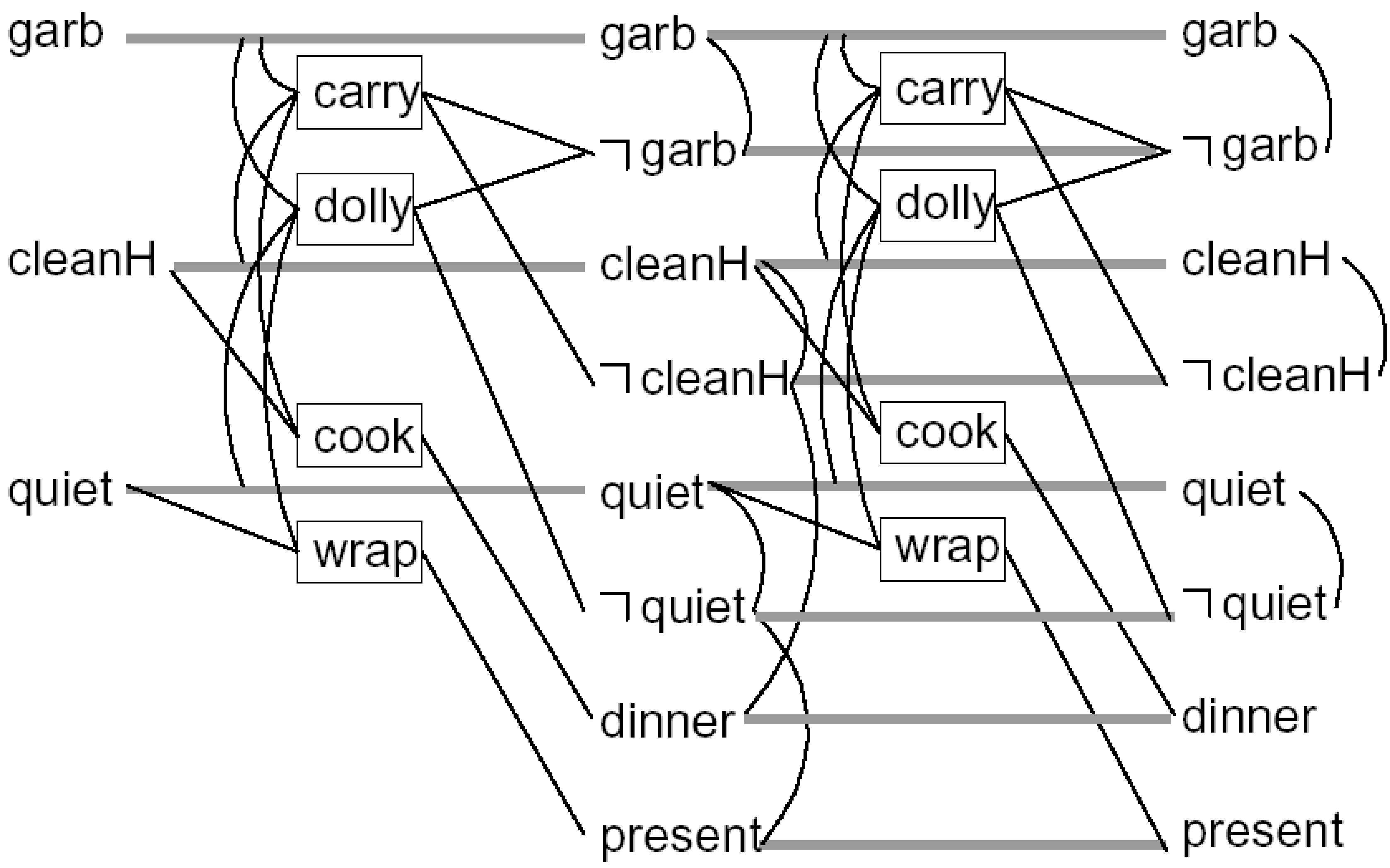
Dinner Date example

- Initial Conditions: (and (garbage) (cleanHands) (quiet))
- Goal: (and (dinner) (present) (not (garbage))
- Actions:
  - Cook :precondition (cleanHands)
    :effect (dinner)
  - Wrap :precondition (quiet)
    :effect (present)
  - Carry :precondition
    :effect (and (not (garbage)) (not (cleanHands))
  - Dolly :precondition
    :effect (and (not (garbage)) (not (quiet)))

Dinner Date example



Dinner Date example



Dinner Date example